

Number Speller

```
?PRINT SPELL 1427
ONE THOUSAND FOUR HUNDRED TWENTY SEVEN
?
```

This program takes a whole number as input and outputs the number spelled out in words.

The general idea is to divide the number into groups of three digits. For example, the number 1234567890 is 1 billion, 234 million, 567 thousand, 890. For each such group we must spell out its three-digit number and also find the word (like “million”) that indicates the position of that group in the entire number.

Spelling a Group of Three

Let’s start by writing a procedure, `SPELL.GROUP`, that spells out a number of up to three digits.

```
TO SPELL.GROUP :GROUP
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST :GROUP "HUNDRED
    SPELL.GROUP BF :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF :GROUP]
IF AND :GROUP>10 :GROUP<20 [OUTPUT TEEN :GROUP-10]
OUTPUT SE (IF :GROUP>9 [TENS FIRST :GROUP] [[]])
    (IF 0<LAST :GROUP [DIGIT LAST :GROUP] [[]])
END
```

```
?PRINT SPELL.GROUP 425
FOUR HUNDRED TWENTY FIVE
?
```

Subprocedures `DIGIT`, `TEEN`, and `TENS` select words corresponding to a particular digit in different positions. `DIGIT` selects words like “three”; `TEEN` words like “thirteen”; and `TENS` words like “thirty.”

The first instruction in `SPELL.GROUP` deals with a nonzero hundreds digit of the group, if any. Next, a possible leading zero is eliminated from the group. Then the procedure recognizes the special case of a number greater than ten and less than twenty. These numbers are special because they are represented all in one word, like “thirteen.” Other two-digit numbers are represented by one word for the tens digit and one for the ones digit, like “eighty seven.” If the number isn’t a teen, the procedure then deals with its tens digit and its ones digit separately.

A trick used in `SPELL.GROUP` looks like this:

```
IF predicate [ expression ] [[]]
```

Here is an example:

```
IF :GROUP>9 [TENS FIRST :GROUP] [[]]
```

If the predicate tested by IF is FALSE, the value of this expression is the empty list ([]), so it contributes nothing to the final result when combined with other things using SE.

SPELL.GROUP outputs the empty list, not the word ZERO, if its input is 0. This is okay because we want to say "zero" only if the entire number we're spelling is 0, not just one group. (Remember that the reason we wrote SPELL.GROUP for numbers up to three digits is that groups of three are the building blocks of larger numbers.) For example, the number 1000234 is spelled "one million two hundred thirty four," not "one million zero thousand two hundred thirty four." We'll have to remember to notice, later on, if the entire number we're spelling is 0.

Here are the procedures that select the words for each digit.

```
TO TENS :DIG
  OUTPUT ITEM :DIG [TEN TWENTY THIRTY FORTY FIFTY
    SIXTY SEVENTY EIGHTY NINETY]
END

TO TEEN :DIG
  OUTPUT ITEM :DIG [ELEVEN TWELVE THIRTEEN FOURTEEN FIFTEEN
    SIXTEEN SEVENTEEN EIGHTEEN NINETEEN]
END

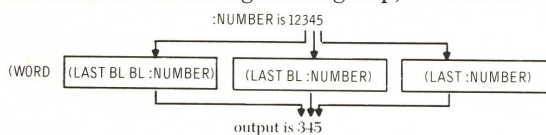
TO DIGIT :DIG
  OUTPUT ITEM :DIG [ONE TWO THREE FOUR FIVE SIX SEVEN
    EIGHT NINE]
END
```

These use the common subprocedure ITEM.

```
TO ITEM :NUM :STUFF
  IF :NUM=1 [OP FIRST :STUFF]
  OP ITEM :NUM-1 BF :STUFF
END
```

Spelling a Large Number

Now we have to divide a large number into groups of three, so that we can use SPELL.GROUP on each of the triads. One complication is that in dealing with very large numbers, we can't rely on Logo's arithmetic operations, because if we do, the numbers will be rounded off. Logo ordinarily handles numbers only up to ten digits without rounding. We'll use Logo's word-manipulation operations. For example, if we're spelling out the number 12345 and want to find the rightmost group, we'll do something like this:



In other words, we must treat a large number as a word that happens to be composed of digits instead of letters.

Note: In order to convince Logo not to round off numbers longer than ten digits, you have to type them in with a quotation mark like this:

```
PRINT SPELL "1234567890987654321
```


WORDPLAY

We can work up from `SPELL.GROUP`. One thing we need is a procedure to combine a spelled-out group with the name of its place in the complete number (thousand, million, etc.):

```
TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP :PLACE]
OP []
END
```

The test for `:GROUP>0` is there to deal with cases like 1000234, where the entire thousands group should be omitted.

At this point, it's important to decide whether we are working on the number from left to right or from right to left. The most obvious thing is probably left to right, because that's the way we actually read numbers, starting with the leftmost group. That's the approach I took the first time I wrote this program. But it turns out to be much simpler to write the program if we start from the right. There are two reasons for this.

The first reason is this: suppose you see a long number like 123,456,234,345,567,678,346,765,654,987. What is the name of the place associated with the leftmost group? To answer that question you have to count the groups, starting from the right. The 987 group is the ones group, the 654 group is the thousands group, the 765 group is the millions group, and so on. So in a sense we have to start from the right in order to know what to do with the 123 group on the left. The second reason is related to the first. Sometimes numbers are written with commas separating the groups. But in Logo we don't use commas inside numbers this way. Suppose you see a number like 1234567890987654321. What is the leftmost group? You might guess 123, but that would be true only if the number of digits in the entire number were a multiple of three. Actually, this number is 1 quintillion 234 quadrillion and so on. In order to know the number of digits in the leftmost group, we have to count off by threes from the right.

Working from right to left, the overall pattern of the program will be more or less like the following. I've written this in lower case to emphasize that it isn't a completed Logo procedure.

```
to spell.number :number
op se (spell.number butlast3 :number) (triad last3 :number)
end
```

Two things are missing from this partially written procedure. First, there is no *stop rule* to tell the procedure when it has reached the end (the leftmost end, that is) of the number. Second, we haven't provided for the place-name input to `TRIAD`. The solution to the first problem is that when the number of digits in the number we're spelling is three or fewer, we're down to the last group. The solution to the second problem involves providing a list of group place names as another input to this partly written procedure. Putting these things together results in two procedures.

```
TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [[] THOUSAND MILLION BILLION TRILLION
QUADRILLION QUINTILLION]
END
```

```

TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF :PLACES)
    (TRIAD LAST3 :NUMBER FIRST :PLACES)
END

```

The top-level procedure, SPELL, recognizes the special case of the number 0. In its subprocedure SPELL1, two auxiliary procedures are used that we haven't written yet. LAST3 and BUTLAST3 are operations like LAST and BUTLAST, but they output (all but) the last three letters of a word instead of (all but) the last one. Here they are:

```

TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL :WORD) (LAST :WORD))
END

```

```

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END

```

SUGGESTIONS

- What do you have to do to make this program spell out numbers in a language other than English? The main thing, of course, is to change the lists of words in SPELL, DIGIT, TENS, and TEEN. But what *structural* differences are there in different languages? For example, in French there are no special names for 70 and 90. Instead, numbers are added to the names for 60 and 80. That is, 70 is "soixante-dix," or "sixty-ten"; 73 is "soixante-treize" or "sixty-thirteen." (This is true of French as spoken in France; the dialect of French spoken in Belgium *does* have special words for 70 and 90!)
- Can you modify the program to spell out numbers including a decimal fraction, so SPELL 3.14 will output [THREE AND FOURTEEN ONE-HUNDREDTHS]? What about exponential notation, so that SPELL 4E3 will output [FOUR THOUSAND]?
- What about translating to or from Roman numerals? In what ways would a program to do that be similar to this one? How would it be different?
- What about translating backward? That is, write a program that will accept a list of words representing a number and output the number.

PROGRAM LISTING

```

TO SPELL.GROUP :GROUP                                :GROUP] [[]] (IF 0<LAST :GROUP ►
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST ►                [DIGIT LAST :GROUP] [[]])
    :GROUP "HUNDRED SPELL.GROUP BF ►                END
    :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF ►                TO TENS :DIG
    :GROUP]                                           OUTPUT ITEM :DIG [TEN TWENTY THIRTY ►
IF AND :GROUP>10 :GROUP<20 [OUTPUT ►                 FORTY FIFTY SIXTY SEVENTY EIGHTY ►
    TEEN :GROUP-10]                                  NINETY]
OUTPUT SE (IF :GROUP>9 [TENS FIRST ►                END

```



```

TO TEEN :DIG
OUTPUT ITEM :DIG [ELEVEN TWELVE ►
    THIRTEEN FOURTEEN FIFTEEN SIXTEEN ►
    SEVENTEEN EIGHTEEN NINETEEN]
END

TO DIGIT :DIG
OUTPUT ITEM :DIG [ONE TWO THREE FOUR ►
    FIVE SIX SEVEN
    EIGHT NINE]
END

TO ITEM :NUM :STUFF
IF :NUM=1 [OP FIRST :STUFF]
OP ITEM :NUM-1 BF :STUFF
END

TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP ►
    :PLACE]
OP []
END

TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [[] THOUSAND MILLION ►
    BILLION TRILLION QUADRILLION ►
    QUINTILLION]
END

TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER ►
    FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF ►
    :PLACES) (TRIAD LAST3 :NUMBER ►
    FIRST :PLACES)
END

TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL ►
    :WORD) (LAST :WORD))
END

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END

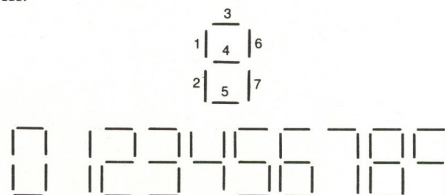
```

Drawing Letters

This project lets the turtle draw letters using a multiple-segment system like that of digital watches. It illustrates Logo's list processing capability and the use of RUN with program-generated Logo instructions. That is, instead of just carrying out procedures that were written ahead of time, this program actually assembles lists of Logo instructions and then carries out those instructions to draw the letters.

Drawing Letters in Segments

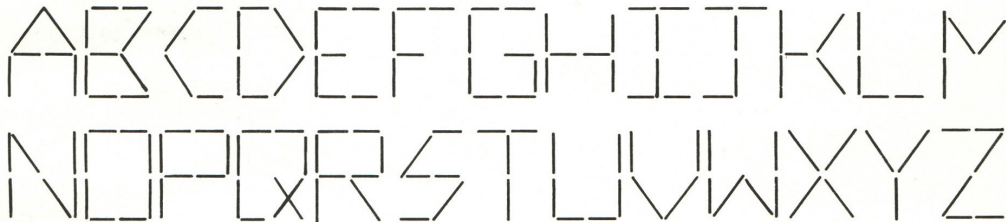
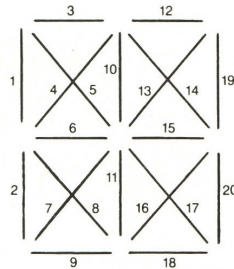
Digital watches, which only have to display digits, generally use a seven-segment system.



Seven-segment display for digits

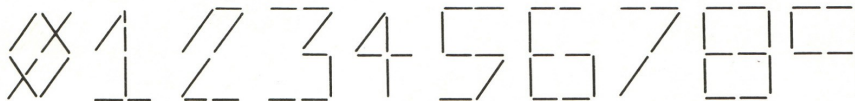
By Brian Harvey

To display all the letters of the alphabet, I chose to use a twenty-segment system, illustrated below.



Twenty-segment display for letters

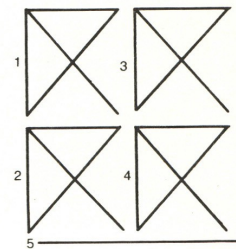
Of course, it would be possible to write a separate procedure for each letter, giving explicit turtle motion commands to shape the letter precisely. The advantage of the segment idea is that it makes it possible to write a single program, then design the individual letters very quickly. For example, after I had finished the letters of the alphabet, it was very easy for me to add the ten digits, even though I hadn't planned for them initially.



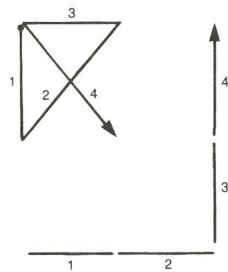
Twenty-segment digits

I could have written twenty procedures, one for each segment. Each would start from a "base" position, move the turtle to one end of the segment, draw the segment, and return to the base position. Then each letter could be described as a list of numbers, identifying the segments that are used to draw the letter. Instead, I chose to try to find some regularities in the way the segments are arranged. I divided the twenty segments into five groups of four each. In each group, the segments can be drawn in a single continuous path, without drawing any segment twice. (I would have liked to be able to draw the entire group of twenty segments continuously without duplication, but that's impossible.) Four of my five groups are identical in shape; the fifth is special.

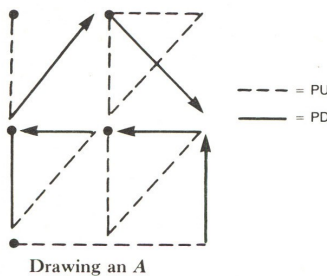
The five groups are numbered in a specific order. Within a group, the segments are also numbered in a specific order; this is shown in the next figure. The program is written so that it draws segments in this order. That is, to draw a letter, the program first draws the four segments that make up the arrow-shaped group in the top left corner. Then the program goes on



Dividing twenty segments into five sets of four each



Order of segments within a group



Drawing an A

to the second group, the arrow-shaped one at the bottom left, and so on. Within each group, the program first draws segment 1, then 2, 3, and 4.

Not all segments are used in every letter, of course. Therefore, the turtle lifts its pen while tracing some of the segments. For example, consider this representation of the letter A.

```
MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
          [PU PU PD] [PU PUPD]]
```

The variable A contains a list of five lists. Each of these smaller lists corresponds to one of the five groups of segments. The first sublist is [PU PD]; this means that the turtle's pen should be up during the first segment and down during the second segment. (There could be up to four words in each sublist. In this case, since there are only two words, the program will stop tracing the first group of segments after the second segment in the group.) This figure shows how the program draws the letter A; compare it to the list just given.

The Letter-Drawing Procedures

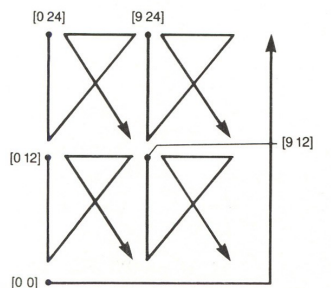
The procedure LETTER draws a letter. It takes two inputs. The first is a list like the one stored in the variable A; the second is a position, that is, a list of two numbers. The letter described by the list is drawn at the position. (Actually it is the lower left corner of the letter that is at the given position.) For example, if we have defined the variable A as just given, we could say

```
LETTER :A [0 0]
```

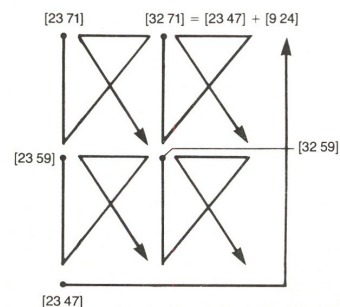
Here is the procedure:

```
TO LETTER :LET :POS
SEGMENTS :LET [ [[0 24] ARROW] [[0 12] ARROW]
                [[9 24] ARROW] [[9 12] ARROW] [[0 0] FINISH] ] :POS
END
```

The LETTER procedure uses a subprocedure SEGMENTS. The second input to SEGMENTS is a list that describes the overall layout of the groups of segments. Like the letter descriptions, it is a list containing five lists. But each of the five lists has only two elements: the starting position of the group of segments and the name of a procedure to draw the group of segments. This procedure is called ARROW for the first four groups and FINISH for the fifth group. The "position" of the beginning of the segment group is actually relative to the position of the letter as a whole, not an absolute screen position. For example, if the position of the letter is [23 47] and the relative position of the third segment group is [9 24], then the actual screen position for that group is [32 71].



Starting points of segments in relative coordinates



Starting points of segments for a letter drawn at POS=[23 47]

To know why the position numbers are what they are, you must know that I chose to base the segment lengths on a 3-4-5 right triangle. The horizontal segments are 9 turtle steps long, the vertical ones 12 steps long, and the diagonal ones 15 steps long. This conveniently makes all the FORWARD commands use whole-number inputs. It is also a reasonable shape for the overall letters.

The procedure SEGMENTS has three inputs. The third is the position of the letter. The first two are both five-element lists of lists. One is a letter description; the other is the overall layout description. The job of SEGMENTS is to match each element of the letter with the corresponding element of the description. It invokes the subprocedure SEGMENT with these sublists as inputs:

```
TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTY? :LET [STOP]
IF NOT EMPTY? FIRST :LET
  [SEGMENT FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END
```

Let's see how this works with a particular example. Suppose we ask Logo to draw the letter A with this instruction:

```
LETTER :A [23 47]
```

This ends up invoking SEGMENTS this way:

```
SEGMENTS [[PU PD] [PD PU PD] [PU PU PU PD]
          [PU PU PD] [PU PU PD]]
  [ [[0 24] ARROW] [[0 12] ARROW]
    [[9 24] ARROW] [[9 12] ARROW] [[0 0] FINISH] ]
  [23 47]
```

Then SEGMENTS invokes SEGMENT five times.

```
SEGMENT [PU PD] [[0 24] ARROW] [23 47]
SEGMENT [PD PU PD] [[0 12] ARROW] [23 47]
SEGMENT [PU PU PU PD] [[9 24] ARROW] [23 47]
SEGMENT [PU PU PD] [[9 12] ARROW] [23 47]
SEGMENT [PU PU PD] [[0 0] FINISH] [23 47]
```

Each element of the list that is specific to the letter A (for example, [PU PD]) is matched with an element of the list that describes the layout of letters in general (for example, [[0 24] ARROW]).

Drawing Each Segment Group

Remember that each sublist of the template (the overall layout description) has two pieces: the relative position of the group and the name of the procedure that draws the group. SEGMENT first has to position the turtle, then invoke the correct procedure. To position the turtle, SEGMENT uses a subprocedure called ADDPOS, which adds two position lists just as we did a few paragraphs ago. Then it uses the RUN command to invoke the procedure ARROW or FINISH, as the case may be. These procedures take the letter

WORDPLAY

description sublist as input, so the procedure name must be linked with that list to form the Logo instruction for RUN.

```
TO SEGMENT :LETPART :TEMPPART :POS
  PU
  SETPOS ADDPOS :POS FIRST :TEMPPART
  RUN LIST LAST :TEMPPART :LETPART
END
```

For example, the first use of SEGMENT in drawing the letter A in our example is

```
SEGMENT [PU PD] [[0 24] ARROW] [23 47]
```

This is equivalent to the following Logo instructions.

```
PU
SETPOS ADDPOS [23 47] [0 24]
RUN LIST "ARROW [PU PD]
```

This is, in turn, equivalent to

```
PU
SETPOS [23 71]
ARROW [PU PD]
```

The tricky (but exciting!) thing to understand here is that the instruction ARROW [PU PD] doesn't actually appear in any Logo procedure in this program. Instead, this instruction is put together as the program is run. SEGMENT combines the word ARROW (which it found in the template list) with the list [PU PD] (which it found in the letter description list) into one big list. It then uses the RUN command to interpret that list as a Logo instruction. We'll use the same trick again later.

The procedures ARROW and FINISH have to follow a certain path, setting the turtle's pen up or down between steps as specified in the letter description. They use a common subprocedure DRAW, which knows how to do that. One of the inputs to DRAW is the letter description sublist with the PU and PD commands; the other input is a list of four Logo instruction lists, one for each segment of the group.

```
TO ARROW :PENS
  DRAW :PENS [[SETH 180 FD 12] [LT 143.13 FD 15]
              [LT 126.87 FD 9] [LT 126.87 FD 15]]
END

TO FINISH :PENS
  DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT 90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
  IF EMPTY? :PENS [STOP]
  RUN FPUT FIRST :PENS FIRST :CMDS
  DRAW BF :PENS BF :CMDS
END
```

Here is how this works out in our example with the letter A. The five invocations of SEGMENT listed earlier result in four invocations of ARROW and one of FINISH.

```
ARROW [PU PD]
ARROW [PD PU PD]
ARROW [PU PU PU PD]
ARROW [PU PU PD]
FINISH [PU PU PD]
```

We'll look at the first invocation of ARROW in more detail. ARROW invokes DRAW like this:

```
DRAW [PU PD] [[SETH 180 FD 12] [LT 143.13 FD 15]
             [LT 126.87 FD 9] [LT 126.87 FD 15]]
```

Just as SEGMENTS paired elements of its list inputs, so does DRAW. It ends up executing these Logo instructions:

```
RUN FPUT "PU [SETH 180 FD 12]
RUN FPUT "PD [LT 143.13 FD 15]
```

There might have been up to four of these RUN instructions, because there are four segments in an ARROW group, but in this case there were only two pen commands in the input list :PENS. If we look at what the RUN instructions actually do in this example, we see that the final effect is just as if the procedure contained these instructions:

```
PU SETH 180 FD 12
PD LT 143.13 FD 15
```

This is a straightforward series of turtle graphics commands. Again, though, it's important to understand that that series of commands is not actually part of any procedure. Instead, the commands were *generated* by the DRAW procedure by putting together pieces of its inputs.

Final Details

Here is ADDPOS, the subprocedure of SEGMENT that turns the relative position of a segment group into an absolute position:

```
TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2
           (LAST :POS1)+LAST :POS2
END
```

Finally, the procedure SAY takes an entire word as input and draws the letters in that word one by one. It's used like this:

```
SAY "HELLO [0 0]
```

and here it is.

```
TO SAY :WORD :POS
IF EMPTY? :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 0]
END
```


WORDPLAY

Here are the definitions for my letters:

```

MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
          [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD PD]
          [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [PU PD] [] [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU PD]
          [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD]
          [PU PU PD] [PU PD PU PD] [PD PU PU PD]]
MAKE "R [[PD PU PD] [PD PU PD] [PU PU PD]
          [PU PU PD PD] [PU PU PU PD]]
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] [PU PD PD] [PD]]
MAKE "T [[PU PU PD] [] [PD PU PD] [PD]]
MAKE "U [[PD] [PD] [] [] [PD PD PD PD]]
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] [PU PU PU PD]]
MAKE "W [[PD] [PD PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "X [[PU PU PU PD] [PU PD] [PU PD] [PU PU PU PD]]
MAKE "Y [[PU PU PU PD] [] [PU PD] [PD]]
MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] [] [PD PD]]

```

SUGGESTIONS

- Make up descriptions for the twenty-segment digits shown near the beginning of this write-up.
- The letter *L* is described very efficiently by this scheme; the turtle takes no unnecessary steps to draw it. The letter *A*, on the other hand, is not very efficiently described. Each *PU* in its description represents a step that the turtle takes without drawing anything; to draw six strokes, the turtle travels over fifteen segments. Can you work out a way to group the segments that makes more letters more efficient? (I don't have any secret answer to this; I haven't tried it myself.)
- Modify the procedures so that the size of the letters can be varied. You could have an input called *SIZE* and use *3* :SIZE* for the horizontal segments, and so forth.
- Modify the procedures so that the aspect ratio of the letters (the ratio of the vertical segment length to the horizontal segment length) is variable. This is much harder; in general, it requires using trigonometry.
- Make up descriptions for lower-case letters. This may require changing the whole arrangement of segments, since some lower-case let-

ters have *descenders*. That is, they extend below the baseline of the capital letters. These letters are *g, j, p, q,* and *y*. Manufacturers of computer terminals don't always use descenders for lower-case letters. Some avoid it by printing those letters higher than they should be; others just use SMALL CAPITALS instead of lower case.

- Without changing the letter descriptions, change the shapes embodied in the procedures ARROW and FINAL. See if you can invent an interesting new alphabet this way.
- Modify the procedures so that you can write words at an angle, not just horizontally across the screen.

PROGRAM LISTING

```

TO LETTER :LET :POS
SEGMENTS :LET [ [[0 24] ARROW] [[0 12] ►
  ARROW] [[9 24] ARROW] [[9 12] ►
  ARROW] [[0 0] FINISH] ] :POS
END

TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTY :LET [STOP]
IF NOT EMPTY FIRST :LET [SEGMENT ►
  FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END

TO SEGMENT :LETPART :TEMPPART :POS
PU
SETPOS ADDPOS :POS FIRST :TEMPPART
RUN LIST LAST :TEMPPART :LETPART
END

TO ARROW :PENS
DRAW :PENS [[SETH 180 FD 12] [LT ►
  143.13 FD 15] [LT 126.87 FD 9] ►
  [LT 126.87 FD 15]]
END

TO FINISH :PENS
DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT ►
  90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
IF EMPTY :PENS [STOP]
RUN FPUT FIRST :PENS FIRST :CMDS
DRAW BF :PENS BF :CMDS
END

TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2 ►
  (LAST :POS1)+LAST :POS2
END

TO SAY :WORD :POS
IF EMPTY :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 0]
END

MAKE "A [[PU PD] [PD PU PD] [PU PU PU ►
  PD] [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD ►
  PD] [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU ►
  PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] ►
  [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU ►
  PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU ►
  PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] ►
  [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] ►
  [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] ►
  [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] ►
  [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU ►
  PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [PU PD] [] ►
  [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU ►
  PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] ►
  [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU ►
  PD] [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD] [PU PU PD] ►
  [PU PD PU PD] [PD PU PU PD]]

```



```

MAKE "R [[PD PU PD] [PD PU PD] [PU PU PD] [PU PU PD PD] [PU PU PD PD]]
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] [PU PD PD] [PD]]
MAKE "T [[PU PU PD] [] [PD PU PD] [PD]]
MAKE "U [[PD] [PD] [] [] [PD PD PD] [PD]]
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] [PU PU PU PD]]

MAKE "W [[PD] [PD PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "X [[PU PU PU PD] [PU PD] [PU PD] [PU PU PU PD]]
MAKE "Y [[PU PU PU PD] [] [PU PD] [PD]]
MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] [] [PD PD]]

```

Mail

When I was a kid in school, my friends and I liked passing notes to each other. It was reflection on this experience that inspired me to write a mail program. In those olden days, the suspense was great as we waited to see if we could send messages from one side of the room to another without getting caught. With this modern method of letting Logo be the mail carrier, students today find different pleasures.

Using the Program

Since Logo has no mail system of its own, I decided to build one. The essential actions are sending and receiving mail. This project is just one example of an electronic mail system. :em.

The program assumes that you have a disk on which daily mail can be saved. For convenience, you should reserve one diskette specifically to hold the messages and the mail program.* To start the program, type MAIL. You will get a screen that looks like this:

```

- - - - - MAIL - - - - -
TYPE S TO SEND MAIL
TYPE R TO READ YOUR MAIL
TYPE A TO READ ALL MAIL
TYPE X TO EXIT
TYPE Q TO SAVE ON DISK
TYPE # TO REINITIALIZE
      THE LIST OF MESSAGES

```

*You may also change the mail program so that you can use a cassette recorder. Then you would save to the cassette instead of to the disk.

Sending Mail

If you want to send mail, type S.

```
WHO IS THE MESSAGE FOR?  
>LAUREN  
WHO IS THE MESSAGE FROM?  
>CYNTHIA  
BEGIN TYPING YOUR MESSAGE.  
PRESS RETURN AFTER EACH TYPED LINE.  
TYPE . ON A SEPARATE LINE TO END.  
>FOR WHAT CRIME WERE SACCO AND  
>VANZETTI PUT TO DEATH?  
>.  
SEND IT? ( Y OR N )
```

First you are asked who the message is for. A prompt (>) appears, and you type in the name of the person to whom you want to send mail. You are then asked who the message is from, and you type in your name. Next you receive instructions. After you type in your message, you are asked if you really want to send it. If you do, you are informed that the message is in the mailbox.

Reading Your Mail

To read your mail, type R.

You are asked to type in your name. Once you do, your messages appear on the screen.

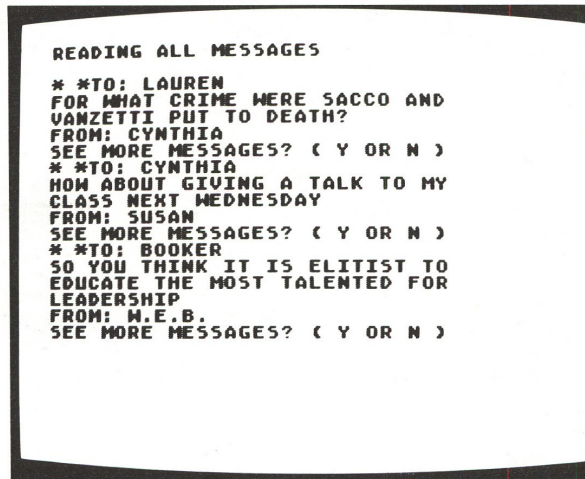
```
TYPE YOUR NAME TO SEE YOUR MESSAGES:  
>LAUREN  
  
LAUREN, HERE ARE YOUR MESSAGES!!!  
  
TO: LAUREN  
FOR WHAT CRIME WERE SACCO AND  
VANZETTI PUT TO DEATH?  
FROM: CYNTHIA  
  
DELETE MESSAGE? ( Y OR N )
```


WORDPLAY

After reading each message, you are asked if you want to delete it. If you type Y, that message is deleted.

Reading All the Mail

If for some reason you want to read all the messages that have been written, type A.



After each message, you are asked if you want to see more messages. If you type Y, you see another message, otherwise you exit from reading all messages.

Other Commands

- Q Automatically saves all messages on the diskette. You are asked if the mail disk is in the drive. If you type Y, the program and all messages are saved on diskette; otherwise the program stops.
- X Stops the program.
- # Deletes all messages.

Structure of the Mail Program

The Data Base

All the messages are organized into one list named ALL.MESSAGES. For example, :ALL.MESSAGES might look like this:

```

[[[TO: JRD] [WHO GOT THE VOTE FIRST: BLACKS OR] [WOMEN]
 [FROM: DAVE]]
[[TO: LAUREN] [FOR WHAT CRIME WERE SACCO AND]
 [VANZETTI PUT TO DEATH] [FROM: CYNTHIA]]
  
```

```

[[TO: LISA][DID THEY EVER DECIDE WHETHER]
[BLACK ENGLISH IS A LANGUAGE OR A DIALECT?]]
[FROM: MARGARET]]
[[TO: JAN] [I HEARD THAT THE BLUEBERRY CROP IN]
[MAINE WILL BE GREAT THIS YEAR.]
[BECAUSE OF THE ACID RAIN.]
[DO YOU BELIEVE IT ?] [FROM: TOM]]
[[TO: BOOKER] [DO YOU AGREE WITH ME THAT THE]
[TALENTED TENTH SHOULD RECEIVE]
[EDUCATION FOR LEADERSHIP] [FROM: W.E.B.]]]

```

Each message is itself a list of lists.

The first message in this example is

```

[[TO: JRD]
[WHO GOT THE VOTE FIRST: BLACKS OR]
[WOMEN]
[FROM: DAVE]]

```

This message contains four sublists:

The first is [TO: JRD].

The second is [WHO GOT THE VOTE FIRST: BLACKS OR].

The third is [WOMEN].

The last is [FROM: DAVE].

The word TO and the receiver's name make up the first list in each message, while the word FROM and the sender's name make up the last list in the message.

The Main Procedure

MAIL is the main procedure of the program. It displays the help text, gets a character command from the user, and checks to see if the command is valid. If it is, it calls the appropriate procedures to carry out the actions. These procedures are SEND.MAIL, MY.MESSAGES, READ.ALL.MAIL, REMOVE.ALL.MESSAGES, and DISK.DUMP.

```

TO MAIL
HELP
MAKE "CHAR RC
IF NOT MEMBERP :CHAR [R S A X Q #] [PR [] PR
  [!!!NOT A COMMAND.]]
IF :CHAR = "R [MY.MESSAGES]
IF :CHAR = "S [SEND.MAIL]
IF :CHAR = "A [READ.ALL.MAIL]
IF :CHAR = "X [STOP]
IF :CHAR = "Q [DISK.DUMP STOP]
IF :CHAR = "#" [REMOVE.ALL.MESSAGES]
PR []
MAIL
END

```

HELP puts the menu of possible actions on the text screen.

WORDPLAY

```

TO HELP
WAIT 50
CT
PR [- - - - - MAIL - - - - -]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END

```

Sending Mail

SEND.MAIL is the main procedure for sending mail. First it asks for the name of the person who is to receive the message. It then asks for your name (the sender). You are then given instructions for typing the message. Finally you are given a chance to change your mind about sending it. If you decide that you want to send it, the message is included in the list of all messages.

```

TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE SE FPUT :ANS GET.MESSAGE
[] LPUT :FROM []
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!!!MESSAGE DELETED!!!!!!] WAIT 50 STOP]
IF EMPTY :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * *]
PR []
END

```

SEND.MAIL uses four subprocedures: RECEIVER'S.NAME, SENDER'S.NAME, GET.MESSAGE, and ADD.THE.MESSAGE.

RECEIVER'S.NAME outputs a sentence of the word TO: and the name of the person who is to receive the message. SENDER'S.NAME works similarly.

```
TO RECEIVER'S.NAME
TYPE ">
OP SE "TO: RL
END
```

```
TO SENDER'S.NAME
TYPE ">
OP SE "FROM: RL
END
```

GET.MESSAGE lets you type in a message, line by line. A "." typed on a separate line signals the completion of the message.

```
TO GET.MESSAGE :MSG
TYPE ">
MAKE "EACH.LINE RL
IF :EACH.LINE = [.] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END
```

In ADD.THE.MESSAGE, the typed message is added to :ALL.MESSAGES.

```
TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT :PRESENT.MESSAGE :ALL.MESSAGES
END
```

Reading Your Mail

MY.MESSAGES is the main procedure for reading your own messages. It gets your name and checks to see if you have any mail by calling CHECK.MY.MESSAGES.

```
TO MY.MESSAGES
CT
IF EMPTY :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR MESSAGES]
TYPE ">
MAKE "ANS RL
IF EMPTY :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [HERE ARE YOUR MESSAGES!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES 0
END
```

CHECK.MY.MESSAGES checks each message in :LIST to see if it is for you. CHECK.MY.MESSAGES takes three inputs. The first is :WHO, the name of the person (you) whose mail it is looking for. The second, :LIST, is the list of messages. The third, :COUNTER, is a message counter that is needed if you should decide to delete a message.

WORDPLAY

```

TO CHECK.MY.MESSAGES :WHO :LIST :COUNTER
PR []
IF EMPTY :LIST [PR [* * * * * THAT'S IT * * * * *]
  STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST FIRST :LIST
  [PRINT.AND.DELETE FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + :COUNTER )
END

```

CHECK.MY.MESSAGES calls PRINT.AND.DELETE, which prints a message and asks if you want to delete it.

```

TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y [MAKE "ALL.MESSAGES DELETE :COUNTER :ALL.MESSAGES
  PR [] PR [!!!!MESSAGE DELETED!!!!]]
END

```

PRINT.AND.DELETE uses PRINT.MESSAGE and DELETE.

PRINT.MESSAGE prints a single message, consisting of the receiver's name, then the message, and finally the sender's name.

```

TO PRINT.MESSAGE :MSG
IF EMPTY :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

```

```

TO DELETE :N :LIST
IF EMPTY :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF :LIST
END

```

Reading All the Mail

The main procedure for reading all the mail is READ.ALL.MAIL; it calls the message-printing procedure, PRINT.ALL.MESSAGES.

```

TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END

```

PRINT.ALL.MESSAGES prints each message and asks you if you want to see more messages.

```

TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTY :ALL [PR [* * * * * NO MORE MESSAGES * * * * *]
  STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * * YOU EXITED MAIL READER * * *]
  STOP]
PRINT.ALL.MESSAGES BF :ALL
END

TO PRINT.MESSAGE :MSG
IF EMPTY :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

```

Saving on the Diskette

DISK.DUMP saves on the disk. First it reminds you to put the disk in the drive.*

```

TO DISK.DUMP
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE????? (Y OR N)]
IF RC = "Y [SAVE "D:MAIL]
END

```

Reinitializing the List of Messages

REMOVE.ALL.MESSAGES clears all messages from the list of messages.

```

TO REMOVE.ALL.MESSAGES
MAKE "ALL.MESSAGES []
END

```

PROGRAM LISTING

TO MAIL	IF :CHAR = "Q [DISK.DUMP STOP]
HELP	IF :CHAR = "# [REMOVE.ALL.MESSAGES]
MAKE "CHAR RC	PR []
IF NOT MEMBERP :CHAR [R S A X Q #] [PR ►	MAIL
[] PR [!!!NOT A COMMAND.]]	END
IF :CHAR = "R [MY.MESSAGES]	
IF :CHAR = "S [SEND.MAIL]	TO REMOVE.ALL.MESSAGES
IF :CHAR = "A [READ.ALL.MAIL]	MAKE "ALL.MESSAGES []
IF :CHAR = "X [STOP]	END

*If you are using a cassette instead of a diskette, you must change the last instruction in DISK.DUMP so that it saves to a cassette: IF RC = "Y [SAVE "C:]


```

TO HELP
WAIT 50
CT
PR [- - - - - MAIL - - - - -]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END

TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED ►
  LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE SE FPUT :ANS ►
  GET.MESSAGE [] LPUT :FROM []
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!!!MESSAGE ►
  DELETED!!!!!!] WAIT 50 STOP]
IF EMPTY :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * ►
  *]
PR []
END

TO RECEIVER'S.NAME
TYPE ">
OP SE "TO: RL
END

TO SENDER'S.NAME
TYPE ">
OP SE "FROM: RL
END

```

```

TO GET.MESSAGE :MSG
TYPE ">
MAKE "EACH.LINE RL
IF :EACH.LINE = [] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END

TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT ►
  :PRESENT.MESSAGE :ALL.MESSAGES
END

TO MY.MESSAGES
CT
IF EMPTY :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR ►
  MESSAGES]
TYPE ">
MAKE "ANS RL
IF EMPTY :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [HERE ARE ►
  YOUR MESSAGES!!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES 0
END

TO CHECK.MY.MESSAGES :WHO :LIST ►
  :COUNTER
PR []
IF EMPTY :LIST [PR [* * * * * ►
  THAT'S IT * * * * *] STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST ►
  FIRST :LIST [PRINT.AND.DELETE ►
  FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + ►
  :COUNTER )
END

TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y [MAKE "ALL.MESSAGES DELETE ►
  :COUNTER :ALL.MESSAGES PR [] PR ►
  [!!!!!!MESSAGE DELETED!!!!!!]
END

```

```

TO DELETE :N :LIST
IF EMPTY :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF ►
:LIST
END

```

```

TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END

```

```

TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTY :ALL [PR [* * * * NO MORE ►
MESSAGES * * * *] STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * * YOU ►
EXITED MAIL READER * * *] STOP]
PRINT.ALL.MESSAGES BF :ALL
END

```

```

TO PRINT.MESSAGE :MSG
IF EMPTY :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

```

```

TO DISK.DUMP
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE????? (Y ►
OR N)]
IF RC = "Y [SAVE "D:MAIL]
END

```

```

MAKE "ALL.MESSAGES [[TO: LAUREN] [FOR ►
WHAT CRIME WERE SACCO AND] ►
[VANZETTI PUT TO DEATH?] [FROM: ►
CYNTHIA]] [[TO: CYNTHIA] [HOW ►
ABOUT GIVING A TALK TO MY] [CLASS ►
NEXT WEDNESDAY] [FROM: SUSAN]] ►
[[TO: BOOKER] [SO YOU THINK IT IS ►
ELITIST TO] [EDUCATE THE MOST ►
TALENTED FOR] [LEADERSHIP] [FROM: ►
W.E.B.]] [[TO: LAUREN] [I HEARD ►
THAT THE BLUEBERRY] [CROP IN ►
MAINE WILL BE GREAT] [NEXT YEAR ►
BECAUSE OF THE ACID RAIN.] [FROM: ►
TOM]] [[TO: TO LISAJ] [IS BLACK ►
ENGLISH CONSIDERED] [A LANGUAGE ►
OR A DIALECT?] [FROM: MARGARET]] ►
[[TO: JRD] [WHO GOT THE VOTE ►
FIRST] [BLACKS OR WOMEN?] [FROM: ►
DAVE]]]

```

Wordscram

A Word Guessing Game

WORDSCRAM picks a word, scrambles the letters, and shows you the scrambled version of the word. Your job is to guess the word. (In this sample game, the word is chosen from a list of thirty or forty technical Logo terms.) WORDSCRAM helps you by showing which letters in your guess are in the correct spot. You can also type `HINT` if you need a hint, or `HELP` if you want to give up. Here is a sample of WORDSCRAM in action.

WORDPLAY**?WORDSCRAM**

WELCOME TO WORDSCRAM !

DO YOU WANT INSTRUCTIONS ? N

THINKING....

OK. HERE IS YOUR SCRAMBLED WORD:

RSNRUCOEI

WHAT'S YOUR GUESS ?

RE

Two letters correct.

WHAT'S YOUR GUESS ?

RE

* *

WHAT'S YOUR GUESS ?

HINT

Get a hint.

WELL...OK...TRY REC

Computer responds.

WHAT'S YOUR GUESS ?

RECUSR

S and R not correct.

****??

WHAT'S YOUR GUESS ?

RECURSION

DOING GREAT !

Got it!

DO YOU WANT ANOTHER WORD ? Y

THINKING....

OK. HERE IS YOUR SCRAMBLED WORD:

PUUTOT

WHAT'S YOUR GUESS ?

HELP

I give up.

THE WORD WAS OUTPUT

DO YOU WANT ANOTHER WORD ? N

Program ends.

Scrambling a Word

The heart of WORDSCRAM is SCRAMBLE. It takes a word as input and outputs a scrambled version of it. The strategy goes something like this. Let's say the word to scramble is "draw."

1. Pick a letter from the word at random.
2. To make sure that the letter does not get picked again, remove it from the word.
3. Join the letter just picked to the result of scrambling the remaining letters of the word. Continue until there are no more letters left.

Using the word "draw" as an example, we might get this result:

```

SCRAMBLE "DRAW
  W + SCRAMBLE "DRA
    R + SCRAMBLE "DA
      A + SCRAMBLE "D
        D + SCRAMBLE "

```

The assembled word is "wrad."

SCRAMBLE picks a letter from the word, then uses that letter in two ways: it removes the letter from the word (to get the input for the recursive invocation of SCRAMBLE), and it sticks the same letter back onto the beginning of the scrambled word. To make this work, after SCRAMBLE picks a letter, it invokes a subprocedure, SCRAMBLE1, whose second input is the letter to remove from the word.

```

TO SCRAMBLE :WORD
IF EMPTY? :WORD [OP "]
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER (SCRAMBLE REMOVE :LETTER :WORD)
END

```

Here is how SCRAMBLE and SCRAMBLE1 interact, in the same example we looked at before.

```

SCRAMBLE "DRAW
  SCRAMBLE1 "DRAW "W
    SCRAMBLE "DRA
      SCRAMBLE1 "DRA "R
        SCRAMBLE "DA
          SCRAMBLE1 "DA "A
            SCRAMBLE "D
              SCRAMBLE1 "D "D
                SCRAMBLE "
                  SCRAMBLE outputs "
                    SCRAMBLE1 outputs "D which is WORD "D "
                      SCRAMBLE outputs "D
                        SCRAMBLE1 outputs "AD which is WORD "A "D
                          SCRAMBLE outputs "AD
                            SCRAMBLE1 outputs "RAD which is WORD "R "AD
                              SCRAMBLE outputs "RAD
                                SCRAMBLE1 outputs "WRAD which is WORD "W "RAD
                                  SCRAMBLE outputs "WRAD

```

Removing a Letter from a Word

REMOVE takes two inputs, a letter and a word. It compares the input letter with each letter of the input word. When it finds a matching letter, it outputs the word with that letter removed.

WORDPLAY

```
?PRINT REMOVE "U "RECURSION
RECRSION
?
```

REMOVE works by comparing the input letter with the first letter of the input word. If they match, then the BUTFIRST of the word is the output we want. Otherwise, the output is formed by joining the first letter of the input word with the result of REMOVEing the input letter from the rest of the word.

```
TO REMOVE :LETTER :WORD
IF :LETTER=FIRST :WORD [OP BF :WORD]      Send back the rest.
OP WORD FIRST :WORD REMOVE :LETTER BF :WORD
END
```

Here is how the preceding example (using RECURSION as the word) happens.

```
REMOVE "U "RECURSION
  REMOVE "U "ECURSION
    REMOVE "U "CURSION
      REMOVE "U "URSION
        REMOVE outputs "RSION
          REMOVE outputs "CRSION which is WORD "C "RSION
            REMOVE outputs "ECRSION which is WORD "E "CRSION
              REMOVE outputs "RECRSION which is WORD "R "ECRSION
```

The remaining procedures in this program are straightforward and won't be explained in detail. You can look at the program listing to see what they are.

SUGGESTIONS

Here are a few ideas for changing WORDSCRAM.

- Change the list of words it knows.
- Tell the player how many guesses it took to get the word.
- After the player guesses the word, ask if she or he would like to see the definition of the word. Since WORDSCRAM's words are technical Logo terms, this would be an interesting way to learn about Logo.
- Add some new messages.
- Do some psychology experiments. Some words look very strange when scrambled. Does this "strangeness" vary from person to person? Some people are better at unscrambling words than others. Why? What sort of strategy do you apply to unscrambling a word? Does it resemble other problem-solving strategies you use?

PROGRAM LISTING

```
TO WORDSCRAM
TS
CT
PR [WELCOME TO WORDSCRAM !]
PR []
PR [DO YOU WANT THE INSTRUCTIONS ?]
IF GETANSWER RC [INSTRUCTIONS] [PR []]
PLAYGAME WIN.MESSAGES GETWORDS
END
```

SEE IF THE USER WANTS INSTRUCTIONS

```
TO GETANSWER :ANS
IF :ANS = "Y [TYPE "YES. OP "TRUE]
IF :ANS = "N [TYPE "NO. OP "FALSE]
PR [PLEASE ANSWER WITH Y OR N]
OP GETANSWER RC
END
```

```
TO INSTRUCTIONS
TS CT
PR (SE [FROM A LIST OF] COUNT GETWORDS [LOGO WORDS, THE])
PR [COMPUTER WILL PICK ONE AT RANDOM AND]
PR [SCRAMBLE IT FOR YOU. YOUR JOB IS]
PR [TO UNSCRAMBLE IT.]
PR []
PR [IT IS NOT NECESSARY TO GUESS THE WORD]
PR [ON THE FIRST TRY. THE COMPUTER WILL]
PR [TELL YOU WHICH LETTERS YOU HAVE IN]
PR [THE RIGHT POSITION BY PRINTING A]
PR [STAR UNDER EACH CORRECT LETTER. A]
PR [LETTER IN THE WRONG POSITION WILL]
PR [HAVE A ? UNDER IT.]
PR []
PR [IF YOU ARE REALLY STUCK, TYPE HINT]
PR [FOR A HINT OR HELP TO SEE THE WORD.]
PR []
PR [GOOD LUCK.]
PR []
TYPE [PRESS ANY KEY TO START...]
MAKE "DUMMY RC
END
```

STARTING THE GAME PLAY

```
TO PLAYGAME :WIN.MESSAGES :WORDS
CT
PR [THINKING ...]
PLAYGAME1 RANPICK :WORDS
IF ANOTHER? [PLAYGAME :WIN.MESSAGES :WORDS] [PR []]
END
```



```

TO PLAYGAME1 :WORD
MAKE "SCRAMBLED SCRAMBLE :WORD
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
MAKE "TOO.MANY.HINTS "FALSE
MAKE "GUESSED.WORDS SE FIRST :WORD []
GET
END

```

SCRAMBLING THE WORD

```

TO SCRAMBLE :WORD
IF :WORD = " [OP "]
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER ( SCRAMBLE REMOVE :LETTER :WORD )
END

TO REMOVE :LETTER :WORD
IF :LETTER = FIRST :WORD [OUTPUT BF :WORD]
OUTPUT WORD FIRST :WORD REMOVE :LETTER BF :WORD
END

```

GETTING THE USER'S GUESS

```

TO GET
PR []
PR [WHAT'S YOUR GUESS ?]
IF ( NOT ( ROW < 23 ) ) [REFRESH.SCREEN]
GETGUESS FIRST RL
END

TO ROW
OP .EXAMINE 171
END

TO REFRESH.SCREEN
SAVE.CURSOR
REDISPLAY
RESTORE.CURSOR
END

TO GETGUESS :GUESS
IF EMPTY :GUESS [OP GETGUESS FIRST RL]
IF :GUESS = "HELP [SHOW.WORD STOP]
IF :GUESS = "HINT [HINT LAST :GUESSED.WORDS GET STOP]
ADDGUESS :GUESS COMPARE :GUESS :WORD
IF :GUESS = :WORD [PR [] PR RANPICK :WIN.MESSAGES ►
STOP] [GET]
END

```

CHECK THE GUESS FOR CORRECT AND INCORRECT LETTERS

```

TO ADDGUESS :GUESS
MAKE "GUESSED.WORDS LPUT :GUESS :GUESSED.WORDS
END

TO COMPARE :GUESS :CORRECT
IF ( OR ( :GUESS = " ) ( :CORRECT = " ) ) [PR [] STOP]
IF ( ( FIRST :GUESS ) = ( FIRST :CORRECT ) ) ►
    [TYPE [*]] [TYPE [?]]
COMPARE BF :GUESS BF :CORRECT
END

```

HINT AND HELP

```

TO HINT :G
IF ( OR ( :G = BL :WORD ) ( :G = ( BL BL :WORD ) ) ) ►
    [MAKE "TOO.MANY.HINTS "TRUE]
IF :TOO.MANY.HINTS [PR [YOU DON'T NEED A HINT!] ►
    PR [THINK SOME MORE.] STOP]
TYPE [WELL]
DODOTS ( 1 + RANDOM 7 )
TYPE [OK]
DODOTS ( 1 + RANDOM 5 )
PR SE [TRY] HINTWORD1 :G :WORD
END

TO HINTWORD1 :W1 :W2
IF EMPTY :W2 [OP []]
IF EMPTY :W1 [OP FIRST :W2]
IF NOT EQUALP FIRST :W1 FIRST :W2 [OP FIRST :W2]
OP WORD FIRST :W2 HINTWORD1 BF :W1 BF :W2
END

TO DODOTS :N
IF :N = 0 [STOP]
WAIT 5
TYPE [.]
DODOTS :N - 1
END

TO SHOW.WORD
PR []
PR SE [THE WORD WAS] :WORD
END

```

MISCELLANEOUS PROCEDURES

```

TO ITEM :N :OBJECT
IF :N = 1 [OUTPUT FIRST :OBJECT]
OUTPUT ITEM :N - 1 BF :OBJECT
END

```



```
TO RESTORE.CURSOR
SETCURSOR :CURSOR
END
```

```
TO REDISPLAY
SETCURSOR [0 0]
PR [THINKING.....]
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
PR []
END
```

```
TO SAVE.CURSOR
PR []
MAKE "CURSOR LIST ( .EXAMINE 172 ) - 1 ( .EXAMINE 171 ) - 1
END
```

```
TO ANOTHER?
PR []
PR [DO YOU WANT ANOTHER WORD ?]
OP GETANSWER RC
END
```

```
TO RANPICK :L
OP ITEM ( 1 + RANDOM COUNT :L ) :L
END
```

```
TO GETWORDS
OP [CATALOG TURTLE FORWARD BACK LEFT RIGHT PROCEDURE ►
    INPUT RECURSION SETBG CIRCLE SQUARE LOGO GRAPHICS ►
    EDIT REPEAT POTS DEFINE COUNT HEADING MEMBERP NODES ►
    PADDLE DYNATURTLE INSTANT BUTFIRST PENDOWN PENUP ►
    PRODUCT RANDOM SETCURSOR SETPC WINDOW TOUCHING MAKE ►
    BUTLAST OUTPUT HIDETURTLE SQRT]
END
```

```
TO WIN.MESSAGES
OP [[HEY, YOU'RE PRETTY SMART] [WHAT A FLUKE!] ►
    [WE ALL GET LUCKY ONCE IN A WHILE!] ►
    [A GOLD STAR FOR YOU] [1 POINT FOR YOU!] ►
    [DOING GREAT!!!] [KEEP UP THE GOOD WORK...]]
END
```

Madlibs™

This project plays the game of Madlibs.* The program asks for words or phrases with which to fill in the blanks in an already-prepared story. Then it prints the resulting story.

*"Madlibs" is a trademark of Price/Stern/Sloan.

By Brian Harvey; story template by Susan Cotten.

Here is an example of a story to be used with the program.

AT _____, _____ WAS _____ING DOWN THE
 time of day person way to move
 STREET. _____ SPOTTED A _____ DIGGING IN A
 person animal
 GARBAGE CAN ACROSS THE STREET. _____ BEGAN TO
 person
 _____ IN THE OPPOSITE DIRECTION BUT IT WAS TOO LATE.
 way to move
 THE _____ SAW _____. IT BEGAN TO CHASE
 animal person
 _____ . _____ TRIPPED AND FELL. THE
 person person
 _____CAME UP BESIDE _____AND BEGAN TO WAG ITS
 animal person
 _____ . _____ REALIZED THERE WAS NOTHING TO
 body part person
 FEAR. _____ REACHED OUT AND PATTED THE _____.
 person animal

Here is what happens when you use the program with this story.

?MADLIB :STORY1

TELL ME A TIME OF DAY

DUSK

TELL ME A PERSON'S NAME

URSULA

TELL ME A WAY TO MOVE

JUMP

TELL ME AN ANIMAL YOU FEAR

RAT

TELL ME A BODY PART

TOE

AT DUSK, URSULA WAS JUMPING DOWN THE STREET.
 URSULA SPOTTED A RAT DIGGING IN A GARBAGE CAN ACROSS
 THE STREET. URSULA BEGAN TO JUMP IN THE OPPOSITE
 DIRECTION BUT IT WAS TOO LATE. THE RAT SAW URSULA. IT
 BEGAN TO CHASE URSULA. URSULA TRIPPED AND FELL. THE
 RAT CAME UP BESIDE URSULA AND BEGAN TO WAG ITS TOE.
 URSULA REALIZED THAT THERE WAS NOTHING TO FEAR. URSULA
 REACHED OUT AND PATTED THE RAT.

?

How a Story Is Represented

A story is represented as a list that contains words and lists (which we'll refer to as *sublists*). The sublists are the blanks of the story. Here is the list that represents the preceding example.

```
MAKE "STORY1 [AT * [HOUR TIME OF DAY]
, [PERSON PERSON'S NAME]
WAS * [MOTION WAY TO MOVE] ING DOWN THE STREET. [PERSON]
SPOTTED A [ANIMAL ANIMAL YOU FEAR] DIGGING IN A GARBAGE CAN
ACROSS THE STREET. [PERSON] BEGAN TO [MOTION] IN THE
OPPOSITE DIRECTION BUT IT WAS TOO LATE. THE [ANIMAL] SAW
* [PERSON] . IT BEGAN TO CHASE * [PERSON] . [PERSON] TRIPPED
AND FELL. THE [ANIMAL] CAME UP BESIDE [PERSON] AND BEGAN
TO WAG ITS * [ANATOMY BODY PART] . [PERSON] REALIZED THERE
WAS NOTHING TO FEAR. [PERSON] REACHED OUT AND PATTED
THE * [ANIMAL] .]
```

Each word or phrase that the user types to replace a blank is given a *name*, so that the program is able to remember it. The named phrase can be used to fill more than one blank. The sublist

```
[MOTION WAY TO MOVE]
```

signals the program to type

```
TELL ME A WAY TO MOVE
```

and to give what the user types the name `MOTION`. Later the sublist `[MOTION]` appears in `:STORY1` without the prompting phrase `WAY TO MOVE`. This signals the program to fill the blank with the word or phrase named `MOTION`, without asking for a new motion.

The Procedures

The top-level procedure is `MADLIB`.

```
TO MADLIB :STORY
PRINT FILL.IN :STORY
END
```

`MADLIB` invokes `FILL.IN` and prints its output, which is a story list with the blanks filled in.

The job of `FILL.IN` is to go through the story list, one element at a time. If an element is a word, that word itself should be part of the output. If the element is a list, it has to fill a blank. Here is the procedure.

```
TO FILL.IN :STORY
IF EMPTY? :STORY [OP []]
IF WORDP FIRST :STORY
  [OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTY? BF FIRST :STORY [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```


This procedure has the overall structure of a recursive operation that does something to every element of a list.

The first instruction is the end test for the input list being empty.

The next line checks for the case in which the first element of the list is a word. In that case, we want to put the word itself in the output.

If the first element isn't a word, it's a blank to be filled. There are two cases. If the list contains more than one word, like [MOTION WAY TO MOVE], that means that the user must be asked for a WAY TO MOVE to fill the blank. The name for what the user types is the first word of the list, MOTION. FILL.BLANK handles this interaction.

```
?SHOW FILL.IN [ [MOTION WAY TO MOVE] QUICKLY! ]
TELL ME A WAY TO MOVE
PERAMBULATE
[PERAMBULATE QUICKLY!]
?
```

If the first element is a list that has only one word, like [MOTION], then we use the word or phrase that was remembered under that name.

```
?SHOW FILL.IN [HELLO, [PERSON NAME];HOW IS [PERSON] TODAY?]
TELL ME A NAME
JONATHAN
[HELLO, JONATHAN ; HOW IS JONATHAN TODAY?]
?
```

The last line of FILL.IN provides the output for both kinds of sublists.

Filling Blanks by Asking Questions

FILL.BLANK has two tasks: it asks the user for a word or phrase, and it gives what the user types a name.

```
TO FILL.BLANK :BLANK
PR SE [TELL ME] ARTICLE BF :BLANK
MAKE FIRST :BLANK RL
END
```

By the way, this is a good example of the use of MAKE with a first input that is not a quoted word. The name of the variable we want to set is part of the story list and does not appear in the text of the procedure.

An elegant detail of FILL.BLANK is that it figures out whether to use A or AN in prompting for a word or phrase. Here is the subprocedure that does the figuring.

```
TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE "AN :PROMPT]
OP SE "A :PROMPT
END
```

```
TO VOWELP :LETTER
OP MEMBERP :LETTER [A E I O U]
END
```

Handling Punctuation

If a blank to be filled is the last thing in a sentence in the story, there is the problem of putting a punctuation mark at the end, without making it a separate word. For example, in our story we have a sentence that ends

```
SAW [PERSON] .
```

If the variable PERSON contains the word URSULA, we'd like the finished story to end

```
SAW URSULA.
```

But if we don't treat this as a special case, the period will be a word by itself:

```
SAW URSULA .
```

The solution I chose is to use an asterisk in the story to mean "take the next two elements in the list and combine them as one word." That's a slight simplification, though, because the next element may be an entire phrase, and only the last word of the phrase can be combined with the punctuation character that follows. The procedure that does the combining is this.

```
TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF :PUNCT
END
```

Here is a revised version of FILL.IN that uses PUNCTUATE.

```
TO FILL.IN :STORY
IF EMPTY :STORY [OP []]
IF EQUALP FIRST :STORY "*" [OP SE (PUNCTUATE FILL.IN
(FPUT FIRST BF :STORY []) FIRST BF BF :STORY)
FILL.IN BF BF BF :STORY]
IF WORDP FIRST :STORY
[OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTY BF FIRST :STORY [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```

PROGRAM LISTING

TO MADLIB :STORY	[FILL.BLANK FIRST :STORY]
PRINT FILL.IN :STORY	OP SE THING FIRST FIRST :STORY FILL.IN ►
END	BF :STORY
	END
TO FILL.IN :STORY	TO FILL.BLANK :BLANK
IF EMPTY :STORY [OP []]	PR SE [TELL ME] ARTICLE BF :BLANK
IF EQUALP FIRST :STORY "*" [OP SE ►	MAKE FIRST :BLANK RL
(PUNCTUATE FILL.IN (FPUT FIRST BF ►	END
:STORY []) FIRST BF BF :STORY) ►	
FILL.IN BF BF BF :STORY]	TO VOWELP :LETTER
IF WORDP FIRST :STORY [OP FPUT FIRST ►	OP MEMBERP :LETTER [A E I O U]
:STORY FILL.IN BF :STORY]	END
IF NOT EMPTY BF FIRST :STORY ►	

```

TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE ►
  "AN :PROMPT]
OP SE "A :PROMPT
END

```

```

TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF ►
  :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF ►
  :PUNCT
END

```

```

MAKE "STORY1 [AT * [HOUR TIME OF DAY] ►
  , [PERSON PERSON'S NAME] WAS * ►

```

```

[MOTION WAY TO MOVE] ING DOWN THE ►
STREET. [PERSON] SPOTTED A ►
[ANIMAL ANIMAL YOU FEAR] DIGGING ►
IN A GARBAGE CAN ACROSS THE ►
STREET. [PERSON] BEGAN TO ►
[MOTION] IN THE OPPOSITE ►
DIRECTION BUT IT WAS TOO LATE. ►
THE [ANIMAL] SAW * [PERSON] . IT ►
BEGAN TO CHASE * [PERSON] . ►
[PERSON] TRIPPED AND FELL. THE ►
[ANIMAL] CAME UP BESIDE [PERSON] ►
AND BEGAN TO WAG ITS * [ANATOMY ►
BODY PART] . [PERSON] REALIZED ►
THERE WAS NOTHING TO FEAR. ►
[PERSON] REACHED OUT AND PATTED ►
THE * [ANIMAL] .]

```
